Usuba, Optimizing & Trustworthy Bitslicing Compiler
Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, Gilles Muller

To cite this version:
Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, Gilles Muller. Usuba, Optimizing & Trustworthy Bitslicing Compiler. WPMVP’18 - Workshop on Programming Models for SIMD/Vector Processing, Feb 2018, Vienna, Austria. 10.1145/3178433.3178437 . hal-01657259v2

HAL Id: hal-01657259
https://hal.archives-ouvertes.fr/hal-01657259v2
Submitted on 16 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract

Bitslicing is a programming technique commonly used in cryptography that consists in implementing a combinational circuit in software. It results in a massively parallel program immune to cache-timing attacks by design.

However, writing a program in bitsliced form requires extreme minutia. This paper introduces Usuba, a synchronous dataflow language producing bitsliced C code. Usuba is both a domain-specific language – providing syntactic support for the implementation of cryptographic algorithms – as well as a domain-specific compiler – taking advantage of well-defined semantics invariants to perform various optimizations before handing the generated code to an (optimizing) C compiler.

On the Data Encryption Standard (DES) algorithm, we show that Usuba outperforms a reference, hand-tuned implementation by 15% (using Intel’s 64 bits general-purpose registers and depending on the underlying C compiler) whilst our implementation also transparently supports modern SIMD extensions (SSE, AVX, AVX-512), other architectures (ARM Neon, IBM Altivec) as well as multicore processors through OpenMP backend.

1 Introduction

Most symmetric cryptographic algorithms rely on S-boxes to implement non-linear (yet reversible) transformations that obscure the relationship between the key and the cyphertext. For example, the third S-box of the Serpent cipher \cite{HugoSerpent} takes 4 bits of input and produces 4 bits of output. To implement this S-box in C, we would typically write:

```c
int SBox3 = { 8, 6, 7, 9, 3, 12, 10, 15, 13, 1, 14, 4, 0, 11, 5, 2 );

int lookup( int index, int[] sbox ) { return sbox[index & 0b1111]; }
```

For example, on input $0110_2 = 6_d$, this function returns the seventh (note the 0-indexing) element of the table, which is $10_2 = 101_0$. This implementation suffers from two problems. First, it wastes register space: a priori, the input will consume a register of at least 32 bits, leaving 28 bits unused, which means that the full potential of the CPU is not exploited. Second, it is vulnerable to cache-timing attacks \cite{VadimCache}: due to cache effects, repeated data-dependent access to a lookup table can leak information on the data being encrypted.

However, it is possible, for example using Karnaugh maps, to compute a logical circuit that implements such a function. The resulting circuit will – by construction – run in constant time regardless of the inputs. For instance, the third S-box of Serpent amounts to the following circuit

```c
node sbox (a,b,c,d:bool) return (w,x,y,z:bool)

vars t01,t02,t03,t04,t05,t06,t07,
t08,t09,t10,t11,t12,t13,t14: bool

let

t01 = a | c; t02 = a ^ b; t03 = d ^ t01;
w = t02 ^ t03; t05 = c ^ w; t06 = b ^ t05;
t07 = b | t05; t08 = t01 & t06; t09 = t03 ^ t07;
t10 = t02 | t09; x = t10 ^ t08; t12 = a | d;
t13 = t09 ^ x; t14 = b ^ t13; z = ~t09;
y = t12 ^ t14;
```

where the four inputs are $a, b, c$ and $d$, the four outputs are $w, x, y$ and $z$, and the remaining variables are temporaries. Seen as a software artefact, it amounts to a function of type $\text{bool}^4 \to \text{bool}^4$: given an input of 4 Booleans, it computes 4 Booleans corresponding to the output of the S-box. For instance, feeding this circuit with $(0, 1, 1, 0) = 0110_2 = 6_d$, will produce the outputs $(1, 0, 1, 0) = 1010_2 = 10_2$, as before. Here, only variables of Boolean type are used, but the same program could manipulate uint64_t values. Conceptually, this would amount to executing 64 such circuits in parallel: one circuit applied to the bits of rank 0 of the registers, another to the bits of rank 1, etc. This works because only bitwise operations are used (AND, OR, XOR and NOT), which means that the bits of each rank of the registers are independently modified.

Bitslicing consists in reducing an algorithm to bitwise operations (AND, OR, XOR, NOT, etc.), at which point we can run the algorithm with bit-level parallelism, viewing a $n$-bits register as $n$ 1-bit registers, and a bitwise AND as $n$-parallel AND operators. To execute such circuit, the inputs must be
converted: the $i$-th bit of the $j$-th input become the $j$-th bit of
the $i$-th register. This operation amounts to a matrix trans-
position transforming $n$ $m$-bit inputs into $m$ $n$-bit registers.

Bitslicing is thus able to increase performance by exploit-
data-parallelism, while improving security by disabling
_cache-timing attacks. Historically, bitslicing is a manual pro-
cess. For instance, here is a snippet of a bitsliced implemen-
tation of the Data Encryption Standard (DES) by Kwan [17]:

\begin{verbatim}
s1 (r31^k[47], r0^k[11], r1^k[26], r2^k[3],
r3^k[13], r4^k[41], &l16, &l12, &l30);
s2 (r3^k[27], r4^k[6], r5^k[54], r6^k[48],
r7^k[19], &l12, &l27, &l1, &l17);
s3 (r7^k[53], r8^k[25], r9^k[33], r10^k[34],
r11^k[17], r12^k[5], &l23, &l15, &l29, &l15);
\end{verbatim}

The full implementation consists of hundreds of lines in the
same, tedious style. Debugging and maintaining such code is
hard at best, and optimizing it even more so. Furthermore,
this code shows its age by failing to exploit modern vector
extensions.

Observing that a bitsliced algorithm is fundamentally a cir-
cuit in software, we designed Usuba, a synchronous dataflow
programming language for implementing circuits
in software. This paper makes the following contributions:

- We have designed Usuba, a synchronous dataflow lan-
guage targeting cryptographical and high-performance
applications. While its syntax and semantics (Section
2) are inspired by Lustre [13], it has been specialized
to address cryptographical needs (lookup tables, per-
mutations, binary tuples, arrays).

- We have implemented a compiler, Usubac, from Usuba
to C (Section 3). It applies standard dataflow compila-
tion techniques [6] as well as domain-specific trans-
formations, such as handling bitvectors, expanding
permutations and lookup tables.

- We have implemented several optimizations (Section
4). This includes standard transformations such as in-
lining, constant folding, common subexpression elimi-
nation (CSE) and copy propagation, as well as a domain-
specific instruction scheduling algorithm. The latter
is tailored to handle the unusual structure of our pro-
grams (large number of live variables, absence of con-
trol structures, for which C compilers have a hard
time generating efficient code.

- We evaluate the end-to-end performance of an Usuba
program, namely DES, on several SIMD architectures
and measure the impact of our optimizations (Sec-
tion 5). At equivalent word size, our Usuba
implementation is 15% faster than a functionally-equivalent,
hand-tuned implementation of DES [17]. Besides, the
same Usuba program can also be compiled to manip-
ulate larger word sizes (such as those offered by the
AVX-512 extensions), yielding a 350% improvement
over the hand-tuned, fixed wordsize implementa-

\begin{verbatim}
\begin{align*}
\langle \text{node} \rangle &::= \text{node } f(\langle \text{tv} \rangle^+) \text{ returns } \langle \text{tv} \rangle^+ \\
&\quad \text{vars } \langle \text{tv} \rangle^+ \quad \text{let } \langle \text{eqs} \rangle \text{ tel} \\
\langle \text{tv} \rangle &::= x : \langle \text{typ} \rangle \\
\langle \text{typ} \rangle &::= \text{bool } | \langle \text{typ} \rangle[n] \\
\langle \text{eqs} \rangle &::= \langle \text{eq} \rangle | \langle \text{eqs} \rangle ; \langle \text{eqs} \rangle \\
\langle \text{eq} \rangle &::= \langle \text{lhs} \rangle = \langle \text{expr} \rangle \\
&\quad | \text{forall } i \in [\langle \text{aexpr} \rangle, (\langle \text{aexpr} \rangle) \langle \text{eqs} \rangle] \\
\langle \text{lhs} \rangle &::= \langle \text{var} \rangle | (\langle \text{var} \rangle \langle \text{var} \rangle^+ ) \\
\langle \text{var} \rangle &::= x | x (\langle \text{aexpr} \rangle)^+ \\
&\quad | x (\langle \text{aexpr} \rangle . . . \langle \text{aexpr} \rangle) \\
\langle \text{expr} \rangle &::= x | n | (\langle \text{expr} \rangle \langle \text{expr} \rangle^+ ) \\
&\quad | \langle \text{unop} \rangle \langle \text{expr} \rangle | \langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle \\
&\quad | \langle \text{fun} \rangle \langle \text{expr} \rangle \\
\langle \text{binop} \rangle &::= + | - | / | \% \\
\langle \text{unop} \rangle &::= \sim \langle \text{binop} \rangle \ ::= \& | || | ^ \\
\langle \text{aexpr} \rangle &::= i | n | \langle \text{aexpr} \rangle \langle \text{binop} \rangle (\langle \text{aexpr} \rangle) \\
\langle \text{fun} \rangle &::= \text{Usuba grammar}
\end{align*}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Usuba grammar}
\end{figure}

Usuba is also trustworthy: we apply translation validation
[21] whereby each run of the compiler is followed by a
verification pass that checks the semantics equivalence
of the source program with the generated code. The same
 mechanism can also be used to validate an optimized Usuba
 program (provided by the user) with respect to a naive
implementation (playing the role of a specification, also pro-
vided by the user). We shall not dwell on these techniques
in the present paper, the description of the formal seman-
tics and the evaluation of the verification framework being
beyond the scope of the workshop.

2 Usuba

Usuba is a synchronous dataflow language akin to a hard-
ware description language (HDL). It enables the concise and
high-level description of software circuits that are then com-
plied into a bitsliced implementation. The benefits of Usuba
are twofold. First and by design, any software circuit spec-
ified in Usuba admits a bitsliced (and therefore efficient)
implementation. Second, the informal description of symmet-
ic cryptographical algorithms (by means of pseudo-circuits,
such as in Menezes et al. [19]) directly translates into Usuba’s
formalism: as a result, one can effectively reason about
and then run the specification. In this section, we present the
syntax and semantics of Usuba.

2.1 Syntax

An Usuba program is composed of a list of node declarations,
permutations and lookup tables.

Nodes correspond to the encapsulating blocks in circuit
 diagrams. A node (Fig. 1, (nd)) specifies its input, output and
lexical typed variables as well as a list of equations (eqs).

Types (typ) can be Booleans or arrays of size $n$ (for a fixed
$n \in \mathbb{N}^*$, known at compile-time). For convenience, we also
offer a type \( u \), which (concisely) stands for \( bool[n] \). An equation (eq) is either a single assignment or a quantified (but statically bounded) definition of a group of equations. A formal variable (var) is either the identifier of a variable, some elements of an array (e.g., \( x[2, 4, 7] \) or even \( x[3] \)) or a contiguous slice of an array (e.g., \( x[2..7] \)). Expressions (expr) are either variables, unsigned integer constants, tuples, unary or binary bitwise operations or node calls. Unary and binary operators (\( unop \) and \( binop \)) correspond exactly to the corresponding C operators. Arithmetic expressions (\( aexpr \)) are evaluated at compile time when a group of equations (defined by a forall) is unfolded into a sequence of standard equations. An arithmetic expression is thus either an unsigned integer constant, a variable introduced by an enclosing forall or an arithmetic operation between two arithmetic expressions.

Permutations are commonly used in cryptographic algorithm to provide diffusion. **Usuba** offers syntactic support for declaring permutations. For instance, the initial permutation of DES amounts to the following declaration specifying which bit of the input bitvector should be routed to the corresponding position of the output bitvector:

```plaintext
perm init_p (input:u64) returns (out:u64) {
  58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28,
  20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6, 64, 56,
  48, 40, 32, 24, 16, 8, 57, 49, 41, 33, 25, 17, 9,
  1, 59, 51, 43, 35, 27, 19, 11, 3, 61, 53, 45, 37,
  29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7
}
```

The direct bitsliced translation of this permutation is a function of 64 Boolean inputs and 64 Boolean outputs, which consists of simple assignments. After copy propagation, a permutation is thus no more than a (static) renaming of variables.

**Look up tables** are, in particular, used to specify S-boxes. The first S-box of DES is thus specified by:

```plaintext
table sbox_1 (input:u6) returns (out:u4) {
  14, 0, 4, 15, 13, 7, 1, 4, 2, 14, 15, 2, 11, 13, 8, 1, 3, 10, 10, 6, 6, 12, 12, 11, 5, 9,
  9, 5, 0, 3, 7, 8, 4, 4, 15, 1, 12, 14, 8, 8, 2, 13, 4, 6, 9, 2, 1, 11, 7, 15, 5, 12, 11,
  9, 3, 7, 14, 3, 10, 10, 0, 5, 6, 0, 13
}
```

The input \( 000111_2 = 7_8 \) produces the output \( 000100_2 = 4_8 \), which is the 8th element of the array. Such a declaration is then converted to a circuit by either looking up a database of known circuits or, failing to find one, compiling it using binary decision diagrams (BDDs).

### 2.2 Static & dynamic semantics

**Usuba** is a statically-typed synchronous dataflow language. Its type and clock system are unsurprising and largely borrowed from earlier synchronous languages, such as Lustre [13]. We refer to Biernacki et al. [6] for a formal as well as algorithmic presentation of type and clock checking. Our treatment of arrays is reminiscent of LustreV4 arrays, which

were coincidently introduced as part of an effort to compile Lustre programs to hardware circuits.

From a semantics standpoint, an USUBA program textually embodies a dataflow graph: an unordered set of equations declaratively describe the “wiring” from input variables to output variables. For instance, the following USUBA programs, which contains the same equations in a different order, are semantically equivalent:

\[
\begin{align*}
x_1 &= \neg a.4; & x_4 &= x_3 \land x_2; \\
x_2 &= \neg a.1; & x_2 &= \neg a.1; \\
x_3 &= a.4 \land a.3; & x_1 &= \neg a.4; \\
x_4 &= x_3 \land x_2; & x_3 &= a.4 \land a.3;
\end{align*}
\]

This equational approach offers referential transparency: any subterm of the program can be hoisted as a new equation with a fresh variable and the subterm replaced by that variable. The compiler backend eventually produces a valid scheduling, which consists in finding an ordering of the equations for which sequential execution of the instructions preserves the intended semantics. These particularities of the dataflow model make it extremely convenient for implementing and proving the correctness of compilation passes: new equations can be introduced at any point, term can be inlined or hoisted, and the scheduling pass eventually takes care of ordering them so as to generate efficient code.

As a result of our design, every USUBA program can be compiled to an efficient bitsliced C program. The resulting imperative code is immune to cache-timing attacks since it cannot perform any input-dependent access to memory.

3 Compilation

This section presents USUBA’s compiler, Usubac, and the generation of C from USUBA. USUBA programs are first simplified to a proper subset of USUBA (Section 3.1). Several optimizations are then applied on the program (Section 4), before generating C code (Section 3.2).

3.1 USUBA0

The high-level constructs offered by USUBA boil down to a strict subset of the language, which we call USUBA0. USUBA0 can be understood as a core calculus for bitsliced algorithms (from a foundational perspective) or as a dataflow assembly language amenable to bitslicing (from a practical perspective). In USUBA0, variables are only of Boolean type. Equations assign a single variable, unless they perform a function call in which case they assign a fully-expanded tuple of Boolean variables. Expressions consist only of bitwise operations and the only constants are of Boolean type, i.e. \(0\) and \(1\). The front-end of the USUBA compiler transforms USUBA programs into (semantically-equivalent) USUBA0 programs by converting lookup tables to boolean circuits, unfolding array definitions, expanding non-Boolean constants, and inlining permutation tables, operators and tuples. The resulting USUBA0 program can then be optimized (Section 4) and eventually compiled to C (Section 3.2).

3.2 Generation of C code

Generating C code from USUBA0 is straightforward, assuming that the equations have been scheduled according to their data-dependencies. Each node is compiled into a function, with return values passed by pointers. Booleans are transformed to either int or vector types (depending on the underlying architecture). Equations are converted into assignments. Expressions are translated to the corresponding C expressions. Function calls are converted to standard C function calls, where the left-hand side of the equation is passed by reference to the function. Generating C code targeting a specific SIMD extension, such as Intel’s SSE, AVX and AVX-512, ARM’s Neon or IBM’s AltiVec, only requires using the bitwise instructions specific to the vector extension, and embedding in the runtime an efficient transposition algorithm adapted to the architecture.

4 Optimizations

Cryptographic applications put high requirements on the performance of the generated code. The C compiler will optimize the code produced by USUBA but our experiments have shown that some optimizations produce better results when done by USUBA’s compiler itself. This is due to the unusual structure of our code: bitsliced programs tend to exhibit hundreds of live variables at any point, which is much more than what C compilers are used to deal with. Optimizing USUBA0 code offers several advantages. Variables are assigned only once and there is no control structure, which means that the code is in single static assignment (SSA) form with no \(\varphi\) node. Moreover, information from the original USUBA program is still available, which can be exploited to guide instruction scheduling for example.

This section describes our optimizations and their impact on the performance of the DES implementation presented in Section 2.1. The C compilers we used are Clang 3.8.0, ICC 17.0.2 and GCC 5.4.0. The measurement were carried on an Ubuntu 16.04 machine with a CPU Intel Core i5-6500 (3.20 GHz). Unless specified otherwise, the flags passed to the C compilers are \(-march=native \neg -O3\). We compare our implementation with Kwan [17]’s bitsliced implementation of DES for 64 bits registers. We refer to it as manual implementation or Kwan’s implementation.

Constant folding Recall that the only operations left in USUBA0 are bitwise operations, function calls and binary constants. Bitwise operations involving constants are simplified, following standard Boolean algebra. Doing this optimization in USUBA produces slightly smaller C files, and guarantees that they are performed, irrespective of the level of optimization at which the C code is compiled.

Common subexpression elimination and copy propagation (CSE-CP) Common subexpression elimination consists in replacing several instances of the same expression by a temporary variable holding the result of this expression. Similarly, copy propagation aims at avoiding useless
with many parameters. Unlike which has a significant impact when dealing with function arguments. The code without inlining is not the small- est, as shown in column "Code size", because it contains the round function and the numerous assignments related to its arguments.

**Scheduling** Usually, C compilers do a good job at scheduling instructions. However, Usuba0’s code has an unusual structure as it is organized in chunks of instructions, each chunk originating from the unfolding of tuples. For example, the Usuba operation $x = a \& b$ where $x$, $a$ and $b$ are of type $u64$ – normalizes to 64 AND instructions, but the 64 results of this computation are unlikely to be all needed at the same time. This causes too many variables to be live at the same time, more than there are available registers, which means that most of them have to be spilled. On the other hand, if those computations were done right before their result is needed, we would reduce its lifespan and thus avoid spilling. Our scheduling algorithm identifies precisely those variables with structurally short lifespans, exploiting the structure of the input Usuba program. For instance, on the naive DES code, the inputs of the S-boxes are computed all at once, but are only used 6 by 6. Therefore, our algorithm schedules those instructions right before they are used, thus removing the need to spill their results.

This algorithm improves performance, as shown in Table 1c: using Usuba’s scheduler instead of relying solely on that of the C compiler increases the performance by 10 to 30% on DES. The C compilers struggle to keep the spilling low: from half to two thirds of the assembly instructions are moves. Using Usuba algorithm reduces the number of moves generated by the C compilers by 19% (Clang) to 43% (GCC).

**Explicit spilling** Conversely, some variables have an inherently long lifespan and will inevitably be spilled. To help the C compiler perform register allocation, we explicitly spill such variables by storing them in arrays. On DES, those variables are typically the outputs of each round, which are potentially not used until the call to the $8^{th}$ S-box of the next round. This reduces the register pressure, and allows the C compiler to find a better register allocation (with less spilling) for the variables exhibiting a shorter lifespan.

Table 1d shows the performance impact of explicitly spilling variables on DES. This optimization heavily depends on the C scheduler and register allocator, which leads to a speedup.

---

### Table 1. Normalized performance impact of the optimization passes

<table>
<thead>
<tr>
<th>CC</th>
<th>without</th>
<th>with</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td>1.23</td>
<td>1.36</td>
<td>$\times 1.11$</td>
</tr>
<tr>
<td>ICC</td>
<td>1.17</td>
<td>1.35</td>
<td>$\times 1.15$</td>
</tr>
<tr>
<td>GCC</td>
<td>1.00</td>
<td>1.07</td>
<td>$\times 1.07$</td>
</tr>
</tbody>
</table>

(a) CSE-CP

<table>
<thead>
<tr>
<th>CC</th>
<th>without</th>
<th>with</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC</td>
<td>1.08</td>
<td>1.41</td>
<td>$\times 1.31$</td>
</tr>
<tr>
<td>Clang</td>
<td>1.12</td>
<td>1.26</td>
<td>$\times 1.13$</td>
</tr>
<tr>
<td>GCC</td>
<td>1.00</td>
<td>1.11</td>
<td>$\times 1.11$</td>
</tr>
</tbody>
</table>

(c) Scheduling

<table>
<thead>
<tr>
<th>CC</th>
<th>without</th>
<th>with</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC</td>
<td>1.26</td>
<td>1.41</td>
<td>$\times 1.12$</td>
</tr>
<tr>
<td>Clang</td>
<td>1.16</td>
<td>1.40</td>
<td>$\times 1.21$</td>
</tr>
<tr>
<td>GCC</td>
<td>1.00</td>
<td>1.01</td>
<td>$\times 1.01$</td>
</tr>
</tbody>
</table>

(d) Explicit spilling

### (b) inlining

<table>
<thead>
<tr>
<th>inlining</th>
<th>speedup</th>
<th>code size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Usuba</td>
<td>$\times 2.14$</td>
<td>91544</td>
</tr>
<tr>
<td>(2) Clang</td>
<td>$\times 1.13$</td>
<td>91576</td>
</tr>
<tr>
<td>(3) without</td>
<td>$\times 1.00$</td>
<td>87480</td>
</tr>
</tbody>
</table>

Usuba WPMVP’18, February 24–28, 2018, Vienna, Austria
varying from 0 to 20% depending on the compiler. The assembly code for DES generated by Clang (resp. ICC) contains 1222 (resp. 1413) fewer move instructions when explicitly spilling some variables, which is directly reflected by the speedup.

5 Evaluation
We evaluate our compiler using our implementation of DES presented in Section 2. While DES is outdated and should not be used for cryptographic purposes, it provides an interesting experimental platform. First, it was not designed with bitslicing in mind, which puts the expressivity of Usuba to the test: our implementation of DES is an almost literal translation of the DES specification, unencumbered by bitslicing details. Second, there exist several, publicly-available implementations of DES, both in standard and bitsliced form for 64 bit architectures. We chose Kwan’s implementation because it is well-known and written in C, enabling us to compare our results across compilers.

5.1 Bitsliced DES without transposition
To isolate the cost of encrypting data from the cost of converting data to a bitsliced format, we first evaluate our implementation of DES without performing transposition of the data. Figure 3a shows the throughput of such an implementation (compiled with ICC 17.0.2 and running an Intel Skylake i9-7900X), depending on the type of registers used, general purpose 64-bit registers (GP-64), or SIMD registers: SSE on 128 bits, AVX on 256 bits, and AVX-512 on 512 bits. We report the throughput of Kwan’s implementation of DES with the label “manual”. For a fair comparison, our implementation uses the same S-boxes as Kwan (whereas smaller versions have been discovered since [20]): we are therefore comparing algorithmically equivalent programs.

The scaling of our code on vector extensions is slightly sublinear. While the C codes of the scalar and vector programs are very close, there are significant differences between the assembly codes: the scalar code is composed of about 12900 instructions and has an arithmetic intensity of 1.85, while the vector codes are composed of about 10400 instructions and have an arithmetic intensity around 4. The main difference between the two codes is the number of move instructions, which directly come from the amount of spilling. This is due to the fact that the vector codes use three-operand instructions while the scalar code only uses destructive two-operand instructions. One might then expect the vector codes to run faster than the scalar code, but this is not the case because Skylake CPUs have 4 scalar ALU ports, and only 3 vector ALU ports. Furthermore, most memory operations can be performed at the same time as some arithmetic operations because they use 3 different ports (2 for loads and 1 for stores). It is worth pointing out that the vector codes are computation bound since their arithmetic intensity is 4, and the vast majority of their memory access are spill-related, and therefore in the L1 cache. Hence, the memory is not the cause of the sublinear scaling on vector extensions. Intel® VTune™ Amplifier reveals that the frontend, and in particular the Micro Instruction Translation Unit (MITE), is limiting the scaling on SSE and even more on AVX. The instruction fetcher can only retrieve 16 bytes of instructions each cycle, which correspond to 4 instructions in the scalar code (whose sizes are around 3.7 bytes) but around 2 or 3 vector instructions, whose sizes are around 4.7 bytes. This is why the vector programs are executed at about 2.6 instructions per cycle (IPC), while the theoretical maximum is above 3.5. The MITE is not an issue in standard vector codes, whose hotspots are small loops (unlike our code, which is fully unrolled), and therefore benefit from the loopback buffer, which can issue up to 4 micro-operations per cycle.
5.2 Transposition

As mentioned in Section 1, in bitslicing, a word of size $m$ must be transposed to occupy 1 bit in $m$ registers, rather than $m$ bits in 1 register, and more generally, $n$ words of size $m$ become $m$ registers of size $n$. We used a standard algorithm to do so [12, 22]. We adapted it to the SIMD extensions we are targeting, which do not support shifts of arbitrary length, but offer special permutation instructions (e.g., blend, vec_perm) that can be used instead of masks and shifts. The complexity of the algorithm is $O(n \log(n))$ for an $n \times n$ matrix. Figure 5 shows the number of cycles needed to transpose 1 bit of input, depending on the register type. The throughput on an Intel Skylake i9-7900X (using code produced by ICC 17.0.2) follows the expected sublinear asymptotic complexity. Similar speedup are obtained with ARM’s Neon and PowerPC’s Altivec extensions.

5.3 Bitsliced DES with transposition

Figure 3b compares the performance of our DES implementations with transposition at various register sizes. For comparison, we also reported the performance of a standard non-bitsliced implementation of DES (from the Crypto++ library [10]). The latter uses general purpose registers of 32 bits mainly, but cannot have its performances increased by using wider registers, unlike the bitsliced versions. Our bitsliced implementation on AVX-512 has a throughput about 15 times higher than a standard DES. The scaling of our implementations is sublinear: $\times 1.81$ from general purpose 64-bit registers to SSE, and $\times 1.75$ from SSE to AVX. This is mainly due to the transposition having sublinear complexity.

Figure 4a shows the speedup obtained by using vector extensions on several architecture: SKL-x is the Intel Skylake i9-7900X aforementioned, KBL is an Intel Kaby Lake i7-7500U, PowerPC is a PPC 970MP, and ARMv7 an ARMv7 Raspberry Pi3. The speedup obtained on KBL and SKL-x are very similar. Using PowerPC’s 128-bit Altivec extensions offers a speedup of 2.36 over the 64-bit general purpose registers, and using ARM’s 128-bit Neon extensions increases performance by a factor 3 compared to the 32-bit registers. Note that the performances for each architecture have been normalized in order to allow a fair comparison of the speedups.

5.4 OpenMP

Another way to increase throughput is to use several cores, which gives (virtually) access to more registers. Usuba can generate code exploiting several cores using OpenMP [9]. We tested the OpenMP code generated by Usuba on a 4x20 cores Intel Xeon E7-8870 v4 (Figure 4b), reporting the speed-up relatively to the moncore, non-OpenMP version. The overhead of OpenMP is negligible while the throughput of DES is almost proportional to the number of cores used.
6 Related Work

**Bitslicing existing algorithms.** Bitslicing was first used in cryptography by Biham [7] on DES. Käser and Schwabe [15] gave a bitsliced implementation of AES, which runs at 7.59 cycles/byte. Their code, unlike most previous bitsliced code, uses only a few variables: their AES state is in only 8 SSE registers, which allows them to use SSE2-specific permutation instructions. Recent work on bitslicing, such as the bitsliced implementations of Prince, LED and Rectangle by Bao et al. [4], have used similar techniques, by representing the cipher’s state on only a few registers instead of dozens or hundreds.

**Bitslicing compilers.** The only bitslicing compiler we know of, bsc, was developed by Pornin [22]. bsc inspired the development of our compiler: we borrowed its presentation of permutation tables and lookup tables. Being a proof-of-concept, the language provided by bsc is less rich than that of Usuba, and the code generated from it is not optimized for speed nor size, whereas our goal is to be more efficient than hand-written code.

**SIMD libraries.** A flurry of C++ libraries provide a unified programming model for various SIMD architectures, such as Boot.SIMD [11], MIPP [8], UME::SIMD [14], Sierra [18] or VC [16]. These works are complementary to ours: our intent was to provide a language that (always) compiles to efficient bitsliced code while being amenable to formal verification. To this end, we have implemented specialized SIMD backends and could certainly benefit from piggy-backing on those libraries, much as we currently use OpenMP for multicore processing.

7 Conclusion & Future Work

Usuba is a synchronous dataflow programming language we have designed to write bitsliced programs. It contains specific constructs to address the specificities of cryptographic algorithms, in particular permutation and lookup tables (Section 2). We presented the compilation steps to get from an Usuba program to an efficient C program (Section 3). The optimizer exploits the dataflow properties of Usuba to make optimizations that most C compilers do not or cannot do as efficiently. We have focused our evaluation on DES but we have also implemented other ciphers such as AES [1], Camellia [3] and Serpent [2].

In future work, we intend to broaden the range of our evaluation to these cryptosystems as well as consolidate our scheduling algorithm across the range of supported architectures.

Acknowledgments

The original idea of a bitslicing compiler was suggested by Xavier Leroy. We are thankful to Intel France and Francois Hannebicq for allowing us to run our experiments on one of their Xeon Skylake Platinum 8168. This work was partially funded by the Émergence(s) program of Paris and the EDITE doctoral school.

References